Design Patterns

The design patterns explored in this chapter are as follows:

• Factory

• Proxy

• Decorator

• Adapter

• Strategy

• State

• Template

• Middleware

• Command

**Factory**

First and foremost, a factory allows us to separate the object creation from its

implementation; essentially, a factory wraps the creation of a new instance, giving

us more flexibility and control in the way we do it. Inside the factory, we can create

a new instance leveraging closures, using a prototype and the new operator, using

Object.create(), or even returning a different instance based on a particular

condition. The consumer of the factory is totally agnostic about how the creation of

the instance is carried out. The truth is that, by using new, we are binding our code

to one specific way of creating an object, while in JavaScript we can have much more

flexibility, almost for free. As a quick example, let's consider a simple factory that

creates an Image object:

function createImage(name) {

return new Image(name);

}

var image = createImage('photo.jpeg');

var image = new Image(name);// javascript way

**A mechanism to enforce encapsulation**

A factory can also be used as an *encapsulation* mechanism, thanks to closures.

*Encapsulation* refers to the technique of controlling the access to

some internal details of an object by preventing the *external* code

from manipulating them directly. The interaction with the object

happens only through its public interface, isolating the external code

from the changes in the implementation details of the object. This

practice is also referred to as *information hiding*. Encapsulation is also

a fundamental principle of object-oriented design, together with

inheritance, polymorphism, and abstraction.

**Proxy**

A **proxy** is an object that controls the access to another object called **subject**.

The proxy and the subject have an identical interface and this allows us to

transparently swap one for the other; in fact, the alternative name for this

pattern is *surrogate*. A proxy intercepts all or some of the operations that are

meant to be executed on the subject, augmenting or complementing their

behavior. The following figure shows the diagrammatic representation:



The preceding figure shows us how the **Proxy** and the **Subject** have the same

interface and how this is totally transparent to the client, who can use one or the

other interchangeably. The Proxy forwards each operation to the subject, enhancing

It's important to observe that we are not talking about proxying

between classes; the Proxy pattern involves wrapping actual

instances of the subject, thus preserving its state.

**Data validation**: The proxy validates the input before forwarding it

to the subject

• **Security**: The proxy verifies that the client is authorized to perform the

operation and it passes the request to the subject only if the outcome of

the check is positive

• **Caching**: The proxy keeps an internal cache so that the operations are

executed on the subject only if the data is not yet present in the cache

• **Lazy initialization**: If the creation of the subject is expensive, the proxy

can delay it to when it's really necessary

• **Logging**: The proxy intercepts the method invocations and the relative

parameters, recoding them as they happen

• **Remote objects**: A proxy can take an object that is located remotely,

and make it appear local

**Decorator**

**Decorator** is a structural pattern that consists of dynamically augmenting the

behavior of an existing object. It's different from classical inheritance, because the

behavior is not added to all the objects of the same class but only to the instances

that are explicitly decorated.



Implementation-wise, it is very similar to the Proxy pattern, but instead of enhancing

or modifying the behavior of the existing interface of an object, it augments it with

new functionalities, as described in the following figure:

In the previous figure, the Decorator object is extending the Component object by

adding the methodC() operation. The existing methods are usually delegated to the

decorated object, without further processing. Of course, if necessary we can easily

combine the Proxy pattern, so that also the calls to the existing methods can be

intercepted and manipulated.

Decorator-levelup-plugin code:

export function levelSubscribe (db) {

    db.subscribe = (pattern, listener) => {

        db.on('put', (key, val) => {

            const match = Object.keys(pattern).every(

                k => (pattern[k] === val[k])

            )

            if (match) {

                listener(key, val)

            }

        })

    }

    return db

}

1. **export function levelSubscribe(db) {**: This line defines and exports a function named **levelSubscribe** that takes a parameter **db**.
2. **db.subscribe = (pattern, listener) => {**: Adds a new method **subscribe** to the **db** object. This method takes two parameters: **pattern** (an object) and **listener** (a function).
3. **db.on('put', (key, val) => {**: Listens for the 'put' event on the **db** object. The 'put' event typically occurs when a new key-value pair is added to the database.
4. **const match = Object.keys(pattern).every(k => (pattern[k] === val[k]));**: Checks if every key in the **pattern** object matches the corresponding value in the incoming **val** object. It uses the **every** method to iterate over the keys and compares them with the values in the **val** object.
5. **if (match) { listener(key, val); }**: If all key-value pairs in the **pattern** match the incoming **val**, it calls the **listener** function with the **key** and **val** parameters.
6. **})**: Closes the 'put' event listener.
7. **}**: Closes the **subscribe** method.
8. **return db;**: Returns the modified **db** object, now with the added **subscribe** method.

In summary, this code extends a given database (**db**) by adding a **subscribe** method. This method allows you to listen for 'put' events and filter them based on a specified **pattern**. If a new key-value pair matches the pattern, the provided **listener** function is called with the key and value.

**Adapter**

The **Adapter** pattern allows us to access the functionality of an object using

a different interface. As the name suggests, it adapts an object so that it can be

used by components expecting a different interface. The following diagram

clarifies the situation:



The preceding diagram shows how the **Adapter** is essentially a wrapper for the

**Adaptee**, exposing a different interface. The diagram also highlight the fact that

the operations of the Adapter can also be a composition of one or more method

invocations on the Adaptee. From an implementation perspective, the most

common technique is composition where the methods of the Adapter provides

a bridge to the methods of the Adaptee. This pattern is pretty straightforward

so let's work immediately on an example.

**Strategy**

The **Strategy** pattern enables an object, called the *Context*, to support variations in

its logic by extracting the *variable* parts into separate, interchangeable objects called

*Strategies*. The context implements the common logic of a family of algorithms, while

a strategy implements the mutable parts, allowing the context to adapt its behavior

depending on different factors such as an input value, a system configuration, or

user preferences. The strategies are usually part of a family of solutions and all of

them implement the same interface, which is the one that is expected by the context.

The following figure shows the situation we just described:



The preceding figure shows how the context object can plug different strategies into

its structure, as they were replaceable parts of a piece of machinery. Imagine a car, its

tires can be considered its strategy to adapt to the different road conditions. We can fit

the winter tires to go on snowy roads thanks to their studs, while we can decide to fit

high- performance tires to go mainly on motorways for a long trip. On the one hand,

we don't want to change the entire car for this to be possible, and on the other, we

don't want a car with eight wheels so that it can go on every possible road.

**Multi-format configuration objects**

Let's consider an object called Config that holds a set of configuration parameters

used by an application, such as the database URL, the listening port of the server,

and so on. The Config object should be able to provide a simple interface to access

these parameters but also a way to import and export the configuration using a

persistent storage, such as a file. We want to be able to support different formats

to store the configuration, as for example, JSON, INI, or YAML.

**State**

**State** is a variation of the Strategy pattern where the strategy changes depending

on the state of the context. We have seen in the previous section how a strategy can

be selected based on different variables such as user preferences, a configuration

parameter, the input provided and once this selection is done, the strategy stays

unchanged for the rest of the lifespan of the context.

In the State pattern instead, the strategy (also called *State* in this circumstance)

is dynamic and can change during the lifetime of the context, thus allowing its

behavior to adapt depending on its internal state, as shown in the following figure:



Imagine that we have a hotel booking system and an object called Reservation that

models a room reservation. This is a classical situation where we have to adapt the

behavior of an object based on its state. Consider the following series of events:

1. When the reservation is initially created, the user can confirm (using

confirm()) the reservation; of course, they cannot cancel (using cancel())

it, because it's still not confirmed. They can however delete (using delete())

it if they change their mind before buying.

2. Once the reservation is confirmed, using the confirm() function again

does not make any sense; however, now it should be possible to cancel

the reservation but not to delete it any longer, because it has to be kept for

the record.

3. On the day before the reservation date, it should not be possible to cancel the

reservation; it's too late for that.

Now, imagine that we have to implement the reservation system that we described

in one monolithic object; we can already picture all the if-else or switch statements

that we would have to write to enable/disable each action depending on the state of

the reservation.

**Template**

The next pattern that we are going to analyze is called **Template** and it also has a

lot in common with the Strategy pattern. *Template* consists of defining an abstract

pseudo class that represents the skeleton of an algorithm where some of its steps are

left undefined. Subclasses can then *fill* the gaps in the algorithm by implementing

the missing steps, called **template methods**. The intent of this pattern is making it

possible to define a family of classes that are all variations of a similar algorithm.

The following UML diagram shows the structure that we just described:



**Middleware**

One of the most distinctive patterns in Node.js is definitely **middleware**.

Unfortunately it's also one of the most confusing for the inexperienced, especially

for developers coming from the enterprise programming world. The reason for the

disorientation is probably connected with the meaning of the term *middleware*, which

in the enterprise architecture's jargon represents the various software suites that

help to abstract lower level mechanisms such as OS APIs, network communications,

memory management, and so on, allowing the developer to focus only on the

business case of the application. In this context, the term *middleware* recalls topics

such as CORBA, Enterprise Service Bus, Spring, JBoss, but in its more generic

meaning it can also define any kind of software layer that acts like a glue between

lower level services and the application (literally the *software in the middle*).

**Middleware in Express**

**Express** (http://expressjs.com) popularized the term middleware in the

Node.js world, binding it to a very specific design pattern. In express, in fact,

a middleware represents a set of services, typically functions, that are organized

in a pipeline and are responsible for processing incoming HTTP requests and

relative responses. An express middleware has the following signature:

function(req, res, next) { ... }

Where req is the incoming HTTP request, res is the response, and next is the

callback to be invoked when the current middleware has completed its tasks

and that in turn triggers the next middleware in the pipeline.

Examples of the tasks carried out by an express middleware are as the following:

• Parsing the body of the request

• Compressing/decompressing requests and responses

• Producing access logs

• Managing sessions

Providing Cross-site Request Forgery (CSRF) protection

If we think about it, these are all tasks that are not strictly related to the main

functionality of an application, rather, they are accessories, components providing

support to the rest of the application and allowing the actual request handlers to focus

only on their main business logic. Essentially, those tasks are *software in the middle*.

**Middleware as a pattern**

The technique used to implement middleware in express is not new; in fact, it can be

considered the Node.js incarnation of the **Intercepting Filter** pattern and the **Chain of**

**Responsibility** pattern. In more generic terms, it also represents a processing **pipeline**,

which reminds us about streams. Today, in Node.js, the word **middleware** is used well

beyond the boundaries of the express framework, and indicates a particular pattern

whereby a set of processing units, filters, and handlers, under the form of functions are

connected to form an asynchronous sequence in order to perform preprocessing and

postprocessing of any kind of data. The main advantage of this pattern is *flexibility*; in

fact, this pattern allows us to obtain a plugin infrastructure with incredibly little effort,

providing an unobtrusive way for extending a system with new filters and handlers.

The following diagram shows the components of the middleware pattern:

